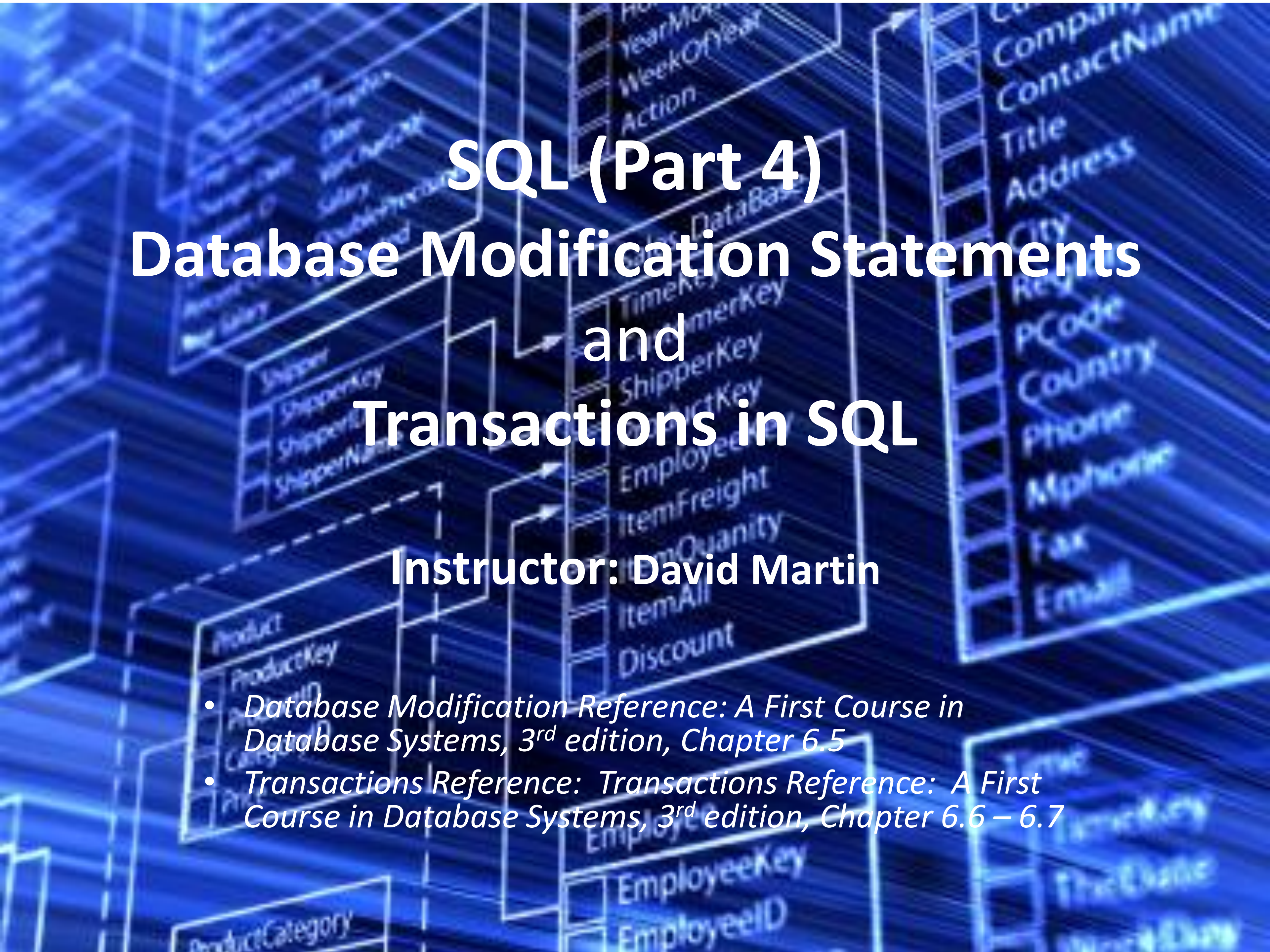


SQL (Part 4)
Database Modification Statements
and
Transactions in SQL

Instructor: David Martin

- *Database Modification Reference: A First Course in Database Systems, 3rd edition, Chapter 6.5*
- *Transactions Reference: Transactions Reference: A First Course in Database Systems, 3rd edition, Chapter 6.6 – 6.7*

Example Tables

Movies

Title	Year	Length	Genre	studioName	producerC#
Pretty Woman	1990	119	Romantic	Disney	999
Monster's Inc.	1990	121	Animation	Dreamworks	223
Jurassic Park	1998	145	Adventure	Disney	675
Star Wars IV	1977	121	Sci-fi	LucasFilm	123

StarsIn

movieTitle	movieYear	starName
Pretty Woman	1990	Julia Roberts
Monster's Inc.	1990	John Goodman

MovieExec

name	address	cert#	netWorth
S. Spielberg	X	675	3000000
G. Lucas	Y	123	4000000
W. Disney	Z	652	5000000

Database Modification Statements

- SQL statements for:
  - *Inserting some* tuples into a relation
  - *Deleting* some tuples from a relation
  - *Updating* values of some columns of some existing tuples

- INSERT, DELETE, and UPDATE are referred to as *modification* operations.
  - They are Data Manipulation Language statements, as is SELECT.

- Modification operations change the *state* of the database.
  - They do not return a collection of rows or other values.
  - They may return errors/error codes.

Insert Statement with Values

```
INSERT INTO R(A_1, …, A_n)
     VALUES (v_1, …, v_n);
```

- A tuple (v_1, …, v_n) is inserted into the relation R, where attribute $A_i = v_i$ and default values are created for all missing attributes.

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
     VALUES (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’);
```

- The tuple (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’) will be added to the relation StarsIn.

```
INSERT INTO StarsIn
     VALUES (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’);
```

INSERT Statement with Query

Movies(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)

```
INSERT INTO Studio(name)
    SELECT DISTINCT studioName
    FROM Movies
    WHERE studioName NOT IN
             (SELECT name
              FROM Studio);
```

- Add to the relation Studio all the names that appear in the studioName column of Movies but do not already occur in the names in the Studio relation.

Semantics of Modifications

- The query must be completely evaluated before any insertion occurs.
  - Why?

- Consider the statement without DISTINCT:

```
INSERT INTO Studio(name)
     SELECT ~~DISTINCT~~ studioName
     FROM Movies
     WHERE studioName NOT IN
               (SELECT name
                FROM Studio);
```

- Database modification statements are completely evaluated on the old state of the database, producing a new state of the database.

DELETE Statement

```
DELECT FROM R
    WHERE <condition>;

DELETE FROM StarsIn
    WHERE movieTitle = ‘The Maltese Falcon’ AND
          movieYear = 1942 AND
          starName = ‘Sydney Greenstreet’;
```

- The tuple (‘The Maltese Falcon’, 1942, ‘Sydney Greenstreet’) will be deleted from the relation StarsIn.

- What if we wanted to delete tuples from StarsIn for <u>all</u> movies starring Sydney Greenstreet?

More DELETE Examples

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

- Deletes all movie executives whose net worth is less than 10 million dollars.

```
DELETE FROM MovieExec
WHERE cert# IN
        (SELECT m.producerC#
         FROM Movies m, StarsIn s
          WHERE  m.title = s.movieTitle AND m.year = s.movieYear
                 AND s.starName = ‘Sydney Greenstreet’);
```

- Deletes all movie executives who produced movies starring Sydney Greenstreet

DELETE: Careful

What does:

```
DELETE FROM MovieExec;
```

without a WHERE clause do?

Answer: Deletes <u>all</u> the tuples from MovieExec

UPDATE Statement

```
UPDATE R
      SET <new-value-assignments>
      WHERE <condition>;
```

- <new-value-assignment> :-

  <attribute> = <expression>, …, <attribute> = <expression>

```
UPDATE Employees
       SET salary = 85000, dept = ‘SALES’
       WHERE SSnum=‘123456789’;

UPDATE Employees
       SET salary = 25000
       WHERE salary IS NULL;

UPDATE Employees
       SET salary = salary * 1.1
       WHERE salary > 100000;
```

UPDATE with Subquery

```
UPDATE R
   SET <new-value-assignments>
   WHERE <condition>;
```

- <new-value-assignment>:-
  <attribute> = <expression>, …, <attribute> = <expression>

```
UPDATE MovieExec
    SET name = 'Pres. ' || name
    WHERE cert# IN ( SELECT presC# FROM Studio );
```

- 2nd line: concatenates the string 'Pres. ' with name.

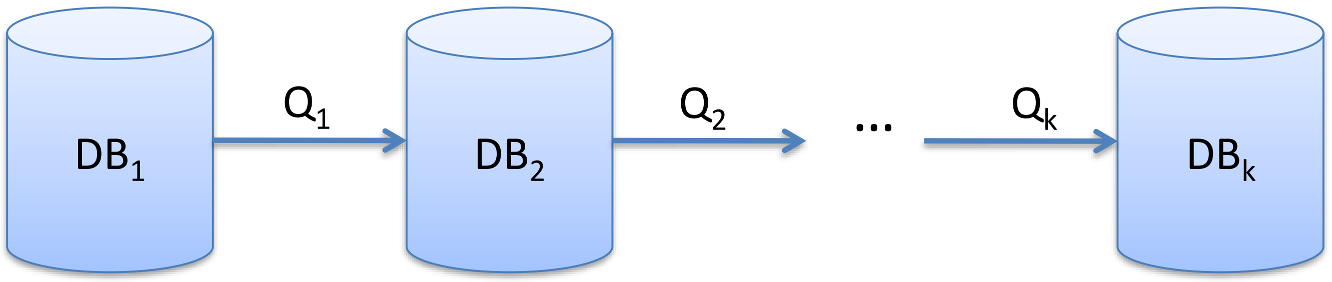
Transactions in SQL

Instructor: David Martin

Reference:
A First Course in Database Systems,
3rd edition, Chapter 6.6 – 6.7

One-Statement-At-a-Time Semantics

- So far, we have learnt how to query and modify the database.
- SQL statements posed to the database system were executed one at a time, retrieving data or changing the data in the database.



Transactions

- Applications such as web services, banking, airline reservations demand high throughput on operations performed on the database.
  - Manage hundreds of sales transactions every second.
  - Transactions often involve multiple SQL statements.
  - Database are transformed to new state based on (multiple statement) transactions, not just single SQL statements.

- It's possible for two operations to simultaneously affect the same bank account or flight, e.g. two spouses doing banking transactions, or an automatic deposit during a withdrawal, or two people reserving the same seat.
  - These "concurrent" operations must be handled carefully.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database constraints preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process survive a crash.
- Optional: weaker forms of transactions are often supported as well.

ACID Transactions (from Wikipedia)

The characteristics of these four properties as defined by Reuter and Härder:

- **Atomicity** requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.
- The **consistency** property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.
- The **isolation** property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method (i.e. if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.
- **Durability** means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Simple Example of What Could Go Wrong

Flights(fltNo, fltDate, seatNo, seatStatus)

- Customer1 issues the following query via a web application.

```
SELECT seatNo
FROM Flights
WHERE fltNo=123 AND fltDate=DATE ‘2012-12-25’ AND
    seatStatus=‘available’;
```

- Customer1 inspects the results and selects a seat, say 22A.

```
UPDATE Flights
SET seatStatus=‘occupied’
WHERE fltNo=123 AND fltDate= DATE ‘2012-12-25’ AND seatNo=‘22A’;
```

Simple Example of What Could Go Wrong (continued)

- Customer2 is also looking at the same flight on the same day simultaneously and decides to choose seat 22A as well.

- PROBLEM SITUATION:

- Both customers believe that they have reserved seat 22A.
- Problem: Each SQL statements of both users is executed correctly, but the overall result is not correct.
- However, a DBMS can provide the illusion that the actions of Customer1 and Customer2 are executed *serially* (i.e., one at a time, with no overlap).
  - **Serializability**

Another Example of What Could Go Wrong, Even with a Single User

Accounts(acctNo, balance)

- User1 wants to transfer $100 from an account with acctNo=123 to an account with acctNo=456.
  1. **Subtract $100 from the account with acctNo=123**

     ```
     UPDATE Accounts
         SET balance = balance – 100
         WHERE acctNo=123;
     ```
  2. **Add $100 to the account with acctNo=456**

     ```
     UPDATE Accounts
         SET balance = balance + 100
         WHERE acctNo=456;
     ```
- What if application or database fails after step 1, but before step 2?

Atomicity

- Failure (e.g., network failure, power failure etc.) could occur after step 1.
  - If this happens, money has been withdrawn from account 123 …
  - … but not not deposited into account 456.

- The DBMS should provide mechanisms to ensure that groups of operations are executed **atomically**.
  - That is, either **all** the operations in the group are executed to completion or **none** of the operations are executed.
  - All-or-nothing, no in-between

Transactions

- A *transaction* is a group of operations that should be executed atomically, all-or-nothing.

- Operations of a transaction can be interleaved with operations of other transactions.

- However, with an “isolation level” called *SERIALIZABLE*, the illusion is given that every transaction is executed one-by-one, in a serial order. The DBMS will execute each transaction in its entirely or not at all.
  - SERIALIZABLE is the default isolation level in SQL
  - Only SERIALIZABLE provides full ACID

Transactions (cont’d)

- START TRANSACTION or BEGIN TRANSACTION (can be implicit)
  - Marks the beginning of a transaction, followed by one or more SQL statements.

- COMMIT
  - Ends the transaction. All changes to the database caused by the SQL statements within the transaction are committed (i.e., they are permanently there--**Durability**) and visible in the database.
  - All changes become visible at once (atomically).
  - Before commit, changes to the database caused by the SQL statements are visible to this transaction, but are not visible to other transactions.

- ROLLBACK
  - Causes the transaction to abort or terminate. Any changes made by SQL statements within the transaction are undone (“rolled back”).

Example Using Informal Syntax

```
BEGIN TRANSACTION
   <SQL statement to check whether bank account 123 has >= $100>
   If there is no account 123, ROLLBACK;
   If account 123 has < $100, ROLLBACK;
   <SQL statement to withdraw $100 from account 123>
   <SQL statement to add $100 to account 456>
   If there is no account 456, ROLLBACK;
COMMIT;
```

- Scenario 1: Suppose bank account 123 has $50.
- Scenario 2: Bank account 123 has $200, bank account 456 has $400.
- Scenario 3: Bank account 123 has $200, bank account 456 has $400, failure after withdrawing $100 from account 123.
- Scenario 4: Bank account 123 has $200, bank account 456 has $400, failure after depositing $100 to account 456, but before COMMIT

Read-Only Transactions

- In the previous examples, each transaction involved a read, then a write.
- If a transaction has only read operations, it is less likely to impact serializability.

- SET TRANSACTION READ ONLY;
  - Stated *before* the transaction begins.
  - Tells the SQL system that the next transaction is read-only.
  - SQL may take advantage of this knowledge to parallelize many read-only transactions.

- SET TRANSACTION READ WRITE;
  - Tells SQL that the next transaction may write data, in addition to read.
  - Default option if not specified; often not specified.

Dirty Reads (Read Uncommitted)

- *Dirty data* refers to data that is written by a transaction but has not yet been committed by the transaction.
- A *dirty read* refers to the read of dirty data written by another transaction.

Pros & Cons:

- **Allow** Dirty Reads
  - More parallelism between transactions.
  - But may cause serious problems, as previous example shows.

- **Don't Allow** Dirty Reads
  - More overhead in the DBMS to prevent dirty reads.
  - Less parallelism, more time is spent on waiting for other transactions to commit or rollback.
  - Cleaner semantics.

Example

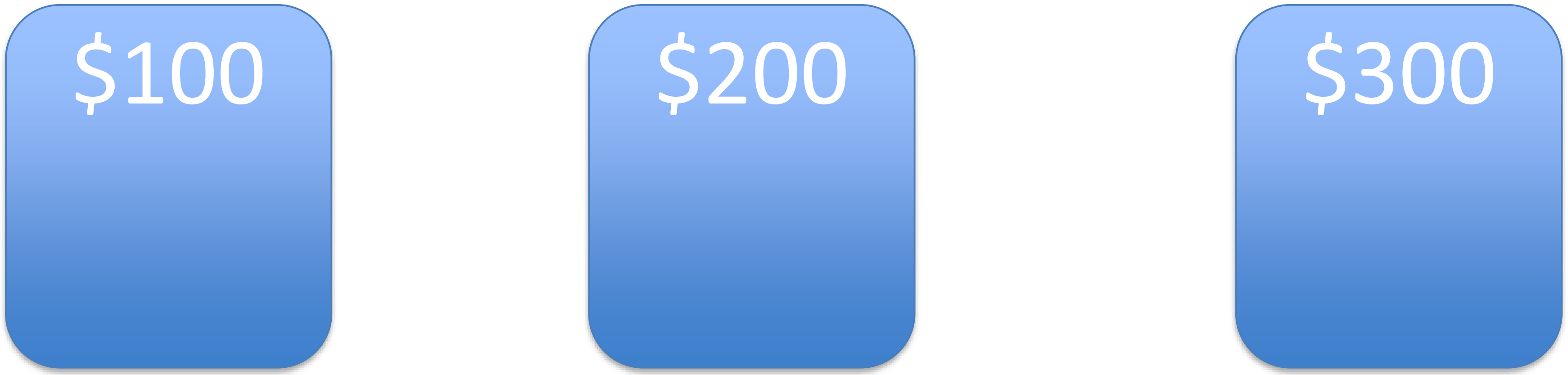
- Consider the following transaction that transfers an amount of money ($X) from one account to another:
  1) Add $X to account 2.
  2) Test if account 1 has $X.
     a) If there is insufficient money, remove $X from account 2.
     b) Otherwise, subtract $X from account 1.

- Transaction T1: Transfers $150 from A1 to A2.
- Transaction T2: Transfers $250 from A2 to A3.
- Initially: A1: $100, A2: $200, A3: $300.
- What might be the (unexpected) result with Dirty Reads if execution of T1 and T2 happens to interlace in a certain way?

Example (cont'd)

- T2: (1) Add \$250 to A3 (now \$550)
- T2: (2) Test passes on A2
- T2: (2b) Subtract \$250 from A2 (now \$100)

- T1: (1) Add \$150 to A2 (now \$350)
- T1: (2) Test fails on A1
- T1: (2a) Subtract \$150 from A2 (now -\$50)



Isolation levels

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

- First line: the transaction may write data (default).
- Second line: the transaction may run with isolation level "read uncommitted".
  - That is, Dirty Reads are allowed.

- Default Isolation Level depends on system.
  - Most systems run with READ COMMITTED or SNAPSHOT ISOLATION

Other Isolation Levels

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
  - Only clean (committed) reads, no dirty reads.
  - But might read data committed by different transactions
    - Might not get same value when you read data a second time in a single transaction

- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  - With repeated queries in a transaction, tuples returned in previous iterations will continue to be returned in later iterations. However, subsequent reads may see phantoms not originally present in the answer.
  - Phantoms are tuples newly inserted while the transaction is running.

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Isolation Levels

Isolation Level	dirty reads	non-repeatable reads	phantoms
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N